(2)

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

THE PROBLEM OF UNDEFINEDNESS IN SPECIFICATIONS

by

Douglas Robert Lengenfelder

June 1988

Thesis Advisor: Daniel Davis

Unclassified

Security Classification of this page

[illegible]

REPORT DOCUMENTATION PAGE

1a Report Security Classification Unclassified	1b Restrictive Markings
2a Security Classification Authority	3 Distribution Availability of Report
2b Declassification/Downgrading Schedule	Approved for public release; distribution is unlimited.
4 Performing Organization Report Number(s)	5 Monitoring Organization Report Number(s)
6a Name of Performing Organization Naval Postgraduate School / 6b Office Symbol *(If Applicable)* 37	7a Name of Monitoring Organization Naval Postgraduate School
6c Address *(city, state, and ZIP code)* Monterey, CA 93943-5000	7b Address *(city, state, and ZIP code)* Monterey, CA 93943-5000
8a Name of Funding/Sponsoring Organization / 8b Office Symbol *(If Applicable)*	9 Procurement Instrument Identification Number
8c Address *(city, state, and ZIP code)*	10 Source of Funding Numbers: Program Element Number / Project No / Task No / Work Unit Accession No

11 Title *(Include Security Classification)* The Problem of Undefinedness in Specifications

12 Personal Author(s) Lengenfelder, Douglas Robert

13a Type of Report	13b Time Covered	14 Date of Report *(year, month,day)*	15 Page Count
Master's Thesis	From To	1988 June 17	55

16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes		
Field	Group	Subgroup

18 Subject Terms *(continue on reverse if necessary and identify by block number)* Term Rewriting Systems, Dendrogrammars, Formal Specification, Algebraic Semamtics, [illegible]

19 Abstract *(continue on reverse if necessary and identify by block number*

Conventional approaches to the formal specifications of computing systems do not provide a facility for leaving elements undefined. The purpose of this thesis is to introduce a formalism for such a facility and to examine its affect on the underlying semantics. These ideas are thus a modification of conventional formalism using semantics. [illegible]

20 Distribution/Availability of Abstract [✓ unclassified/unlimited] same as report DTIC users	21 Abstract Security Classification Unclassified
22a Name of Responsible Individual Davis, Daniel	22b Telephone *(Include Area code)* (408) 646-2174 / 22c Office Symbol 52vv

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted — security classification of this page

All other editions are obsolete — Unclassified

The Problem of Undefinedness in Specifications

by

Douglas Robert Lengenfelder
Captain, United States Air Force
B.S., United States Air Force Academy, 1979

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1988

Author: [illegible]
Douglas Robert Lengenfelder

Approved by: Daniel Davis
Daniel Davis, Thesis Advisor

[illegible]
Yuh-Jeng Lee, Second Reader

[illegible]
Robert B. McGhee, Acting Chairman, Department of Computer Science

[illegible]
James M. Fremgen, Acting Dean of Information and Policy Sciences

ABSTRACT

Conventional approaches to the formal specifications of computing systems do not provide a facility for leaving elements undefined. The purpose of this thesis is to introduce a formalism for such a facility and to examine its affect on the underlying semantics. These ideas are thus a modification of conventional formalism using algebraic semantics.

TABEL OF CONTENTS

I. INTRODUCTION

Within software engineering, there are several models to aid and direct the life-cycle software development of programs. Each model has some type of phase where program specification must occur. With the complexity of today's problems, one would prefer to increase the amount of automation involved in each software engineering phase. This thesis will present a procedure toward aiding the automation of the software engineering specification phase. It is an extension of previous work accomplished at the Naval Postgraduate School in the area of automation (Ref. 1, 2, 3, & 4). Previous work toward automating the specification phase produced a realization of the difficulty in formally handling errors in algebraic specifications (Ref. 5). Instead of only handling error conditions in a specification, this thesis presents a procedure to deal with both error conditions and a broader class of conditions which will be called "undefined conditions." Error conditions are defined as conditions which should not occur within an implementation of a specification. On the other hand, undefined conditions are defined as both error conditions and what we will call "don't care" conditions. "Don't care" conditions are conditions which, if implemented (note that they need not be implemented) do not affect correctness of the implementation.

This new concept of "undefined conditions" (or "undefined objects") in a specification reduces some problems that have occurred with specifying only error conditions, but still requires a method to identify and define what is an "undefined condition." In order to understand the problems involved with error conditions in specifications; formal specifications, syntax and semantics will be discussed in the next two chapters.

II. FORMAL SPECIFICATIONS

A desire for automation within the specification phase of software engineering influences the choice of specification methodology. We chose formal specifications because of the desire for rigor and the ability to use associated mathematical concepts in automating the specification phase.

The concept of formality within a specification, in computer science, is "...devoted to, or done in accordance with, forms or rules...." (Ref. 6) But the intent of formality is deeper than just the rules when discussing specifications. Formality not only deals with the rules of the specification, but also creates a strict methodology of developing the specification. In this sense, formal specifications will have "...rigorously defined syntax and semantics...." (Ref. 7)

A formal specification is therefore defined as "...a specification that is written entirely in a language with an explicitly and precisely defined syntax and semantics." (Ref. 8) Through formalism, one develops an in-depth understanding of the problem being dealt with and furthermore, formalism allows logic and mathematics to be applied more readily toward solving or analyzing a problem. This formalism then leads to the development of algorithms and techniques which are translated to the computer and therefore automated.

Some of the following are salient features which a formal specification should include:

1) we must be able to check whether the specification is consistent
2) the methodology should be implementation independent
3) the specification language should be simple, clear and easy to use

4) the specification should be faithful to the intent of the specifier (MacLennan describes this as Preservation of Information - a principle of programming languages (Ref. 9).

5) the specification language must mirror the problem complexity in terms of expressibility (i.e., a simple object in the program should be simple to describe in the specification language - a concept of cost regularity in terms of expressing programming objects (Ref. 10).

With the above features, formal specifications provide several benefits to the software engineering process. The first advantage results from the specification being implementation independent. This abstraction allows multiple implementations of the same concept with only one specification. Not only does this reduce the amount of effort required for implementing several programs in different environments, but also provides a coherent data base upon which one can provide maintenance to that "family" of programs. The process of transforming a mental image of a program into a specification and then into a family of individual programs if depicted in Figure 1 (Ref 11).

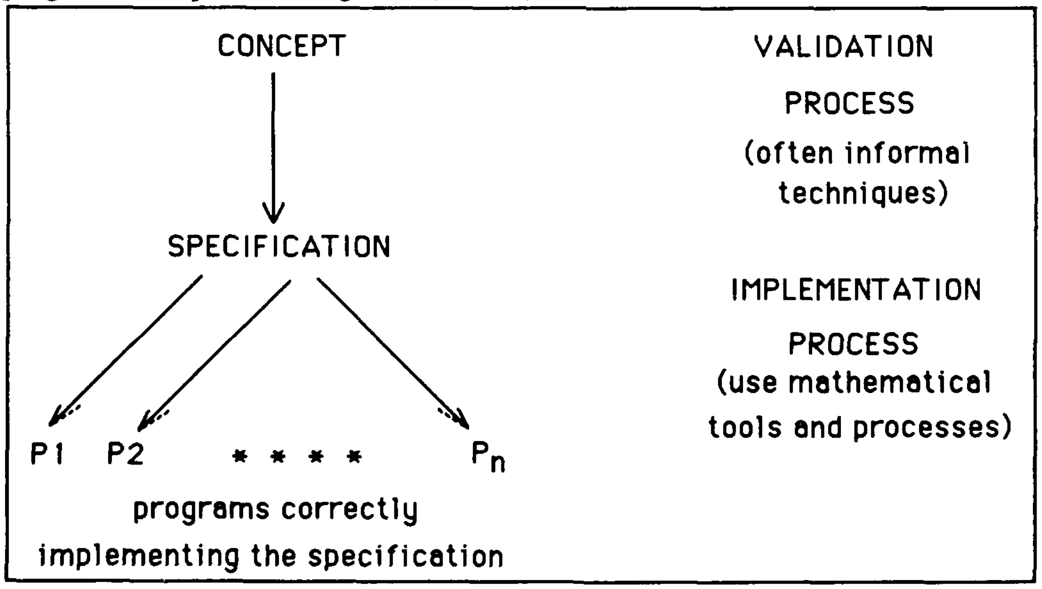

CONCEPT
SPECIFICATION
P1 P2 * * * * Pn
programs correctly implementing the specification
VALIDATION PROCESS (often informal techniques)
IMPLEMENTATION PROCESS (use mathematical tools and processes)

Figure 1. The Development of a Concept into an Implementation of the Concept.

A second advantage of a formal specification is its ability to reduce ambiguity. The specification should describe the essence of what is desired from the implementation. With precise syntax and semantics, formal specifications have less chance for error due to an implementation misunderstanding or misinterpretation. Thus, the formality of the specification language reduces the equivocality that can be encountered in other specification techniques.

A third advantage is the fact that language (and specification) formality provide a basis for rigor in the construction of the specification (Ref. 12). Rigor and reduction of ambiguity reduces errors when implementing a family of programs. Consequently, this early elimination of error is capable of:

1) monetary savings in the software engineering process
2) time savings in the software engineering process
3) improved relationship in the final product

According to Faibian (Ref. 13), the cost of correcting an error increases a factor of approximately 2.5 times for each stage of the software engineering process that the error goes undetected (see Figure 2). If the specification deals with hardware rather than software, this factor increases to approximately four and increases up to a factor of 100 for embedded systems (Ref. 14). Thus, the potential for monetary savings due to reduced errors in the specification stage of software engineering is significant.

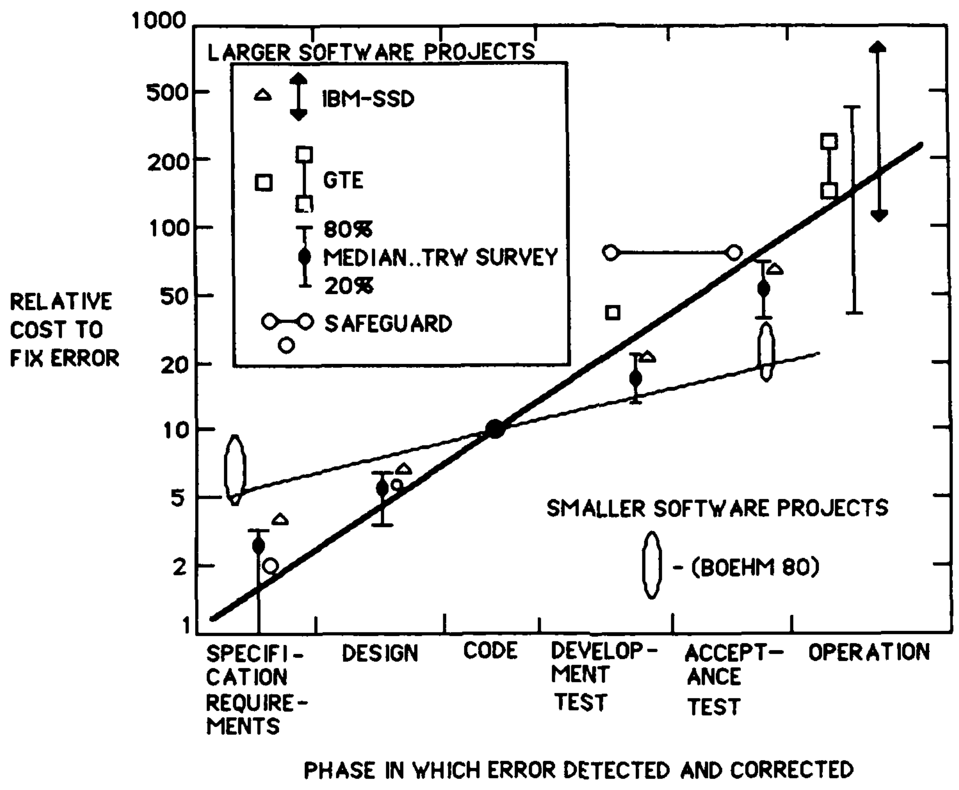


Figure 2. Cost of Errors in Software Engineering Cycle (Ref. 13)

A fourth and foremost advantage of formal specification is automation. Automated processes (such as proof of correctness programs) can check for syntactic correctness and semantic consistency. This in turn provides evidence to the degree of which the specification is well formed and error free. Furthermore, either the implementation process could (theoretically) be automated or the process of checking the consistency of the implementation to the specification could be automated.

However, formal specifications have some problem areas which counteract their benefits (Ref. 15):

1) they require a great deal of mathematical background and can therefore be difficult for the layman to read and understand
2) they can be very expensive to produce (one of the reasons a "family" of programs is hopefully developed from one specification)
3) they can be more verbose than the program itself

The bottom line is that we know no other way to accomplish automation of the specification phase without formal specifications and since automation is our final goal, a formal specification approach becomes an ipso facto choice for the requirements phase in software engineering. Recalling that a rigorous formal specification requires rigorously defined syntax and semantics, the next chapter will deal with syntax and semantics.

III. SYNTAX AND SEMANTICS

Within the computer world, a specific execution of a given program is defined through the use of semantics. Thus, semantics defines how input values are related to output values. Beckman (Ref. 16)

A formal specification can be divided into three definitive areas: 1) syntax , 2) semantics, and 3) pragmatics. Pragmatics deals with implementation time dependencies of specification operators and will not be discussed further because of its irrelevance toward this thesis (we are not trying to dismiss the importance of this area, but only state that pragmatics will not directly effect the proposed procedure). Secondly, since rigorously defined syntax has a great deal of literature describing the process, the following section concentrates on rigorously defining the semantics of a formal specification (which will include the use of syntax).

As stated above by Beckman, semantics defines how input values are related to output values. Thus, input values are literally given meaning by our understanding or notions of what they mean. These notions are further defined through the operations that are allowed to occur on input values. This development of "acceptable" or "desirable" operations on input values to produce our desired output values literally defines the semantics of a program. Thus, the assignment of meaning is "arbitrary" within each individual in the sense that one individual's mental concept of a real world object can be constructed and stored completely different from the next individual's concept. For example, one individual's concept of a bench may be another individual's concept of a chair. (Could a bench not substitute for a chair and if so, can it not be classified as a chair?) A more thorough understanding of this problem of semantics can be achieved through an understanding of the human process of abstracting meaning within the field of computer science.

A. SYNTACTIC REPRESENTATION

In order to understand the process of abstraction in computer science the two concepts of form and meaning must be defined. The first concept to be defined is the concept of form, also known as the syntactic domain or the world of expressions. It is the set of all syntactic expressions. These expressions are identified as either correct or incorrect through well formed formulae. The correctness of the expressions is therefore simply a matter of following rules in constructing well formed formulae and parallels the underlying concepts of parsers in computer science. For example, "chair" is an acceptable expression in the syntactic domain of expressions when considering the English language because the word is found in the English dictionary. Note that no meaning has been placed on the word chair; the fact that it is recognized as an expression is acceptable. Similarly, when a computer program is parsed, syntactic errors are pointed out to the programmer. The parser is not stating that the meaning of the program is wrong because the parser is not concerned with meaning. Instead, the parser is saying it cannot recognize the structure of the program (in terms of control flow, variables, etc).

B. SEMANTIC REPRESENTATION

The second concept requiring definition is the world of meaning, also known as the semantic domain. Within the ordinary world of reality, objects have meaning just because they exist. However, within the computer science field, there is no simple method to handle semantics. But, just as the syntactic domain is handled similar to human language, the semantic domain is also handled similar to a human language. In a dictionary, the word "chair" is given meaning by describing the object in terms of other real world concepts or objects. The dictionary is literally using pointers to point toward other objects in order to derive a meaning for the object being defined. These pointers do not always have to be other physical objects, but when we interpret the dictionary, we must have some base knowledge in order to understand the pointers.

How then does the computer science field implement a "dictionary?" There are three approaches toward a formal semantic description. They include Axiomatic, Operational, and Denotational Semantics.

C. AXIOMATIC SEMANTICS

The axiomatic approach describes the meaning of programs by describing the logical properties of each linguistic feature in the language. Through this process the logical properties of a program can be inferred (Ref. 17). For example, R.W. Floyd (Ref. 18) presents a series of axioms that must be true in order for a desired program to be satisfactory (or faithful to the "intent" of the program). These axioms then describe the semantics or "deep meaning" of the program. Axiomatic specifications have the advantage of ease in accomplishing proofs (whether you are proving correctness of the complete program, logical properties, or only a sub-program or function). However, this specification is very difficult to implement upon a specific real machine.

D. OPERATIONAL SEMANTICS

The operational approach toward semantics in formal specifications describes program meaning by specifying algorithms for translating any specification into an executable process on a hypothetical machine (Ref. 17). This approach is called "operational" because it was meant for ease of implementation. To implement this system, one would first construct a run time system that transforms the real processor into the hypothetical processor. Then, it is just a matter of changing the specification into executable code on the hypothetical processor. One should immediately note that problems of efficiency would be a concern and more importantly, this methodology does not allow an individual to easily reason about properties of programs. (The language PL/I has been defined in the ANSI standard by means of the operational approach. (Ref. 19))

E. DENOTATIONAL AND ALGEBRAIC SEMANTICS

Finally, the denotational approach describes the meaning of a program by describing how to construct mathematical objects from the syntax which in turn denote the meaning of a program (Ref. 17). Thus, denotational semantics assigns meaning to programs by pointing to something that the program denotes. Although this has not worked very well with imperative languages, it has met with success in functional programming such as LISP (Ref. 17). A major problem area with denotational semantics (and functional languages) is that "time" is very difficult to describe within the specification. Advantages include: 1) it parallels our present concept of abstract data types, and 2) it subsumes algebraic semantics, therefore simplifying reasoning about program specifications. Algebraic semantics, a form of denotational semantics, will be the basis for our methodology in assigning meaning to objects. For a more thorough introduction on denotational semantics, see Tennent, R.D., "The Denotational Semantics of Programming Languages," Communication ACM Vol 19, No 8 1976.

Underlying the denotational theory is the belief that mathematical primitive concepts have a clear meaning and if these concepts are then pointed to, the meaning of the program will then become clear (or at least unambiguous). A problem with this approach is the same problem underlying the English "chair" example. That is, without some base knowledge of the English language one can never hope to use the dictionary nor acquire meaning from the objects which the dictionary points to in defining "chair." Similarly, without an extensive knowledge of mathematical concepts one can never expect to be able to derive meaning from the objects pointed to by denotational semantics.

F. PROBLEM AREAS IN DENOTATIONAL AND ALGEBRAIC SEMANTICS

As an understanding of the problem of assigning meaning to objects within the syntactic domain is unveiled, a deeper, more profound problem arises. A concrete object within the semantic domain can be an abstract concept -- for example, a stack can be a real world object in a semantic domain which depicts an abstraction of the FILO (first-in-last-out) concept. If an abstract object is allowed in the semantic domain, then the object may be abstracted and a hierarchy of abstractions are created. Each level of abstraction requires another level of indirection to find a pointer providing definition of the object (using the denotational definition of semantics).

Furthermore, applying a semantic definition to abstract objects is not equally easy for all objects. Some objects (such as love) may be very difficult to point at, while others (such as a basketball) can be considerably easier to point to In addition to the above problems, what may appear to be easy to semantically define may be very difficult to semantically implement. For example, can a large bean bag be a "chair" or is a "chair" really just an abstract concept which defines a complete semantic definition. These problem areas directly impact our ability to determine whether we can provide a pointer to all objects within a syntactic world which gives them meaning. This is not stating that semantic problems are intractable, but instead it states that we cannot determine at the outset whether the problem we are working on is always tractable. The question boils down to: How does one approach the problem of providing meaning to the abstract concepts in the syntactic domain in a consistent manner which preserves the essence of our real world?

We have elected to assign meaning within the syntactic domain through an interpretation. An interpretation of a syntactic object is defined to be the process of pointing to some specific object within the semantic domain (which could be considered the syntactic co-domain in this case). One can think of an interpretation as a mapping of the syntactic domain into the semantic domain. The attributes of the semantic object are therefore transferred to the syntactic object which infers a meaning onto the syntactic object. However, when implementing this mapping, one must deal with conditions/objects such as "undefined terms." "Undefined terms" are objects within the syntactic domain which require no interpretation to an actual object in the semantic domain (i.e., they need not map to any object within the semantic domain). These concepts are depicted in Figure 3.

However, the problem is actually worse than stated above. What is normally done when humans abstract reality (creating a semantic domain within their understanding of the world), is to create a model that is a partial realization of reality. Note that this model is not a projection of the semantic domain (world of reality) nor is the semantic domain a projection onto the model (as the latter would disagree with most concepts of models). Figure 4 describes this improved depiction of how we implement an interpretation to achieve the assignment of meaning to syntactic objects.

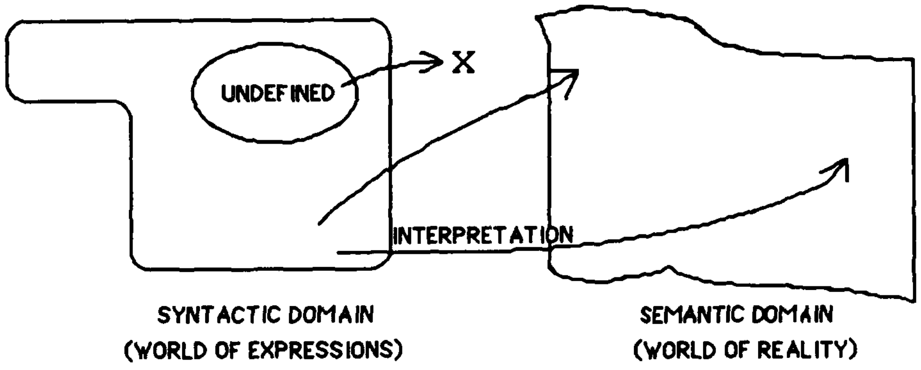


Figure 3. Assigning Meaning to the Syntactic Domain

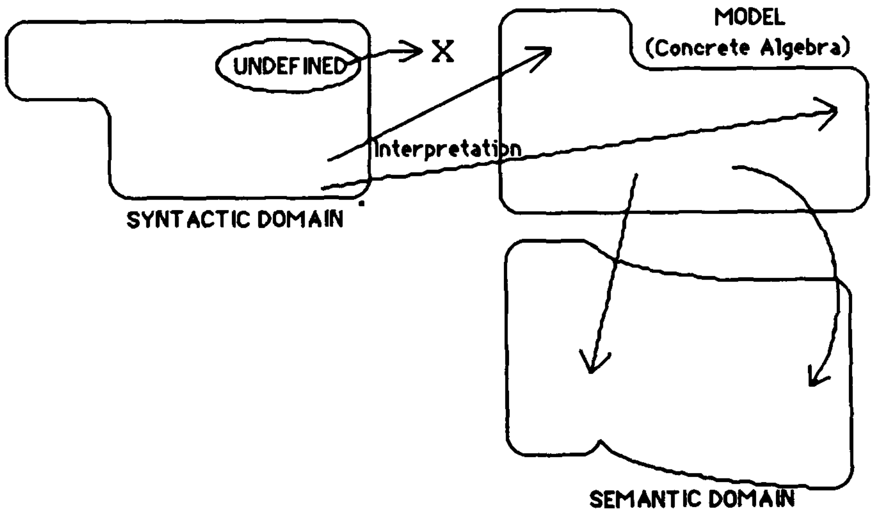


Figure 4. Assigning Meaning to the Syntactic Domain
Through a Model (Concrete Algebra)

For example, if "chair" is the syntactic object, possible interpretations could point to real world objects such as "bean bag chairs," "sofas," "bar stools," etc. If we use the following rule to confirm whether the interpretation is correct: "a chair is any object which is used for the purpose of sitting," then the above interpretations are all correct. One could argue that this is not a very satisfying interpretation for our intuitive notion of the object "chair" but this certainly displays the difficulty in trying to apply a meaning to a syntactic object.

On the other hand, if we use the following to confirm whether the interpretation is correct: "a chair is any object used for only one person to sit on," we would have to remove "sofas" from the above interpretations. This may be a more satisfying interpretation of our notion of "chair."

Another problem displayed in Figure 4 deals with the mapping between the model and the semantic domain. The model is a partial realization in defining the semantic domain, just as our notion of the integers within a computer is only a partial realization of our mathematical understanding of integers (there can only be a finite number of integers represented within the computer, yet axioms are based on the ability of integers to be infinitely large). Thus, the model is predestined to fail in capturing the complete true essence of the semantic domain. Problems of interpreting the modeling process will not be analyzed here, but if the task of creating software is to be approached, then a methodology to approach the problem of mapping the syntactic domain into the model must be developed. This mapping function will be defined as an interpretation. One of the most perplexing problems with developing a consistent methodology toward an interpretation (and certainly a stumbling block for software engineering) is the problem of handling the undefined terms found within the syntactic domain. One proposed approach is to not even allow undefined terms in the syntactic domain.

G. AN APPROACH TOWARD HANDLING ERROR CONDITIONS

Goguen did not allow "don't care conditions" by explicitly defining all "undefined terms" as errors. The errors were then individually described within the specification (Ref. 20). Thus, he required each object within the syntactic domain of a specification to be either 1) a representation of an object/concept or 2) an actual error condition. This creates a total function when mapping a syntactic representation into its semantic meaning (using denotational semantics). Not only did this cause combinatorial explosion problems (since every syntactic object must be defined), but it can lead to inconsistencies within the specification axioms.

Goguen carefully provides strict definition to every possible term to prevent his "error" terms from being considered undefined terms. However, Davis contends that many undefined conditions within the specification are "do not care" situations and can create a consistent specification and implementation whether or not each particular "undefined" condition is considered an error or not (Ref. 21). Goguen's approach, as stated earlier, causes not only combinatorial explosion, but can also create inconsistences within the axiomatic understanding of the interpretation. The inconsistencies occur because several exceptions must now handle each "error" case within the syntactic domain. On the other hand, Davis' approach to undefined conditions is more general than error conditions because undefined conditions includes both "error" and "don't care" conditions.

Whether one prefers defining these conditions as "undefined" conditions or "error" conditions one is still concerned with identifying the undefined terms within the syntactic domain in order to achieve a precise and consistent implementation of their respective model. If one could identify these undefined terms, then any implementation of those terms would create consistent axiomatic systems (just as we have different, but consistent, axiomatic geometries or set theories). However, conceptualizing the individual "undefined conditions" as a class of terms (as depicted in Figure 4) has some problems as will be discussed in the next chapter.

IV. THE PROBLEM OF IDENTIFYING UNDEFINED TERMS

One of our goals is to reduce the combinatorial explosion problem of dealing with individual "error conditions" within a specification. By creating "undefined conditions" we effectively reduce the number of "error conditions" which we must deal with (or prove something about) in a specification. We can further reduce the combinatorial problem by treating all undefined terms as a single class of terms within the specification. Therefore, even though undefined terms may look very different, they can be logically handled in a single consistent manner within a proof of the specification. For example, although x/y and 4x/4y may look different, the second expression reduces to x/y and therefore is semantically equivalent (even though it is not syntactically equivalent) to the first term of x/y. Both of these terms should be semantically handled the same within a specification. Thus, it would be beneficial to equate these two expressions together into one equivalence class and just deal with the equivalence class.

If we are to equate undefined terms for the purpose of identification then we are creating an equivalence class of undefined conditions/syntactic objects. However, to define an equivalence class, one must have a notion of equality. Remember that semantically, the notion of equality (or semantic meaning) is derived from having terms within the syntactic world map into the model (an interpretation) which consequently points to the same concrete object in the semantic domain. All terms that map into the same semantic domain object then create their own equivalence class. Those items which do not map to an object representation within the model then create the "undefined" equivalence class.

To understand whether two items are equal or not, one must be able to determine whether there are any undefined terms that are a part of that term since such a part, makes the term undefined also. But note that the concept of equality is used to define the "undefined" equivalence class. And since the equivalence classes are used in providing meaning to concepts such as equality (through the use of an interpretation), a circular reasoning problem has arisen.

A more insidious problem actually occurs with the above circular reasoning problem. This problem can best be displayed by an arithmetic example. First, we define an undefined term as "any term that has a division by zero." Next, we want to know whether $xy = y$. In the analysis we divide both sides by y and get $x = 1$. If we erroneously conclude that $x = 1$ is the only solution, we have not considered all of the underlying assumptions with our algebraic system. Note that if y equals zero, then x can equal anything, and the equality still holds. This problem occurs because we have used an "undefined term" in solving the equality (the undefined term was the division by y when y equals zero). Thus, the solution is exclusively $x = 1$ if and only if y does not equal zero. Thus, we must not only look for undefined subterms but also for any undefined subterm that appears within any process of proving two terms equal. This considerably complicates the circular reasoning problem and may account for a programmar's inability to identify and properly implement a completely consistent model of a specification.

In order to break the circular chain of definition, we will go back to the original concept of identifying undefined terms within the syntactic domain. Thus the process proposed is to identify undefined terms within formal specifications by identifying syntactic terms which are undefined. To accomplish this, a grammar will be used to syntactically identify objects which lead to undefined terms. This grammar will then create an equivalence class of undefined terms which will point toward the concept of "undefined objects." The implementer of the specification can then select whether he/she desires to implement this class of "undefined objects" as either "errors" or as "don't care" situations (remembering that "don't care" situations could include anything from a system crash to a system call which simply ignores an undefined term).

V. METHODOLOGY

In the next section, "Identifying Undefined Objects," we show we can identify a class of undefined terms within a specific syntactic world. The example is a specification for stacks. Stacks are extremely useful within computer science and yet relatively simple data structures; thus, they appear to be a logical beginning for showing how we can identify "undefined objects" within a specification.

A. THE PROOF PROCESS

As previously discussed, many specifications have functions which create undefined objects. For example, if we deal with a specification for real numbers, the division operator is certainly an accepted operator. However, if we attempt to divide a real number by zero, an undefined term results. This idea of undefined terms is also found in a stack specification.

We first identify undefined conditions within the stack specification. Then, undefined terms are analyzed in order to develop a grammar which can identify all "undefined conditions" within the syntactic domain. Thus, the concept of undefinedness is being identified at the syntactic level from which we can unambiguously deal with "undefined objects" of the specification. These syntactic terms, identified by a grammar, will then become the "don't care" situations within the specification. In other words, once we have identified a class of "undefined objects" within the specification, they will be given semantic meaning by pointing them (denotational semantics) toward the semantic world (real world meaning) of "I don't care how you implement this class of objects." Thus, the implementer of the specification is free to handle this class of "don't cares" in any consistent manner which he/she so desires. The critical step then becomes:

> Can one unequivocally say that the grammar which identifies this class of "undefined objects" truly identifies all "undefined objects" as stated in the specification?

This is the purpose of Chapter 6 - "Identifying Undefined Objects." In order to accomplish this aim, we need to introduce the use of dendrogrammars.

B. DENDROGRAMMARS

Initially a textual grammar was used in proving the capability to semantically identify undefined objects within a formal specification. When this grammar caused confusion with the subtle problems underlying the proof, the author decides to use a dendrogrammar. Tree structures are a common data structure within computer science. Their utility is derived from not only a hierarchical structure (to include nested or one-to-many relationships) but also the regularity involved with the algorithms that manipulate these structures. By using a dendrogrammar, the author hopes that the tree structured grammar provides the algorithmic clarity that the problem requires.

Note that there is no difference between a textual grammar and its equivalent dendrogrammar. In fact, this is a fundamental concept underlying the theory of parsing. For example, if we have a sentence defined by a context-free grammar, parsing that sentence is the same as constructing its syntax tree, starting at the leaf nodes (reference Figures 5 and 6). However, the additional benefit of the tree representation, is that it provides structural information in a clear and precise manner.

An additional advantage of dendrogrammars when dealing with semantics is the fact that the structure of a concept can provide more information (and therefore more meaning) to the reader toward understanding a concept. For example, the following sentence is ambiguous as stated:

They are shooting birds.

One does not know whether a person is pointing at fowl named "shooting birds" or whether there is a group of hunters ahead.

But a tree structure can readily clarify the meaning of the sentence as observed in Figures 5 and 6. This is due to additional information that the tree structure is capable of providing through structural meaning.

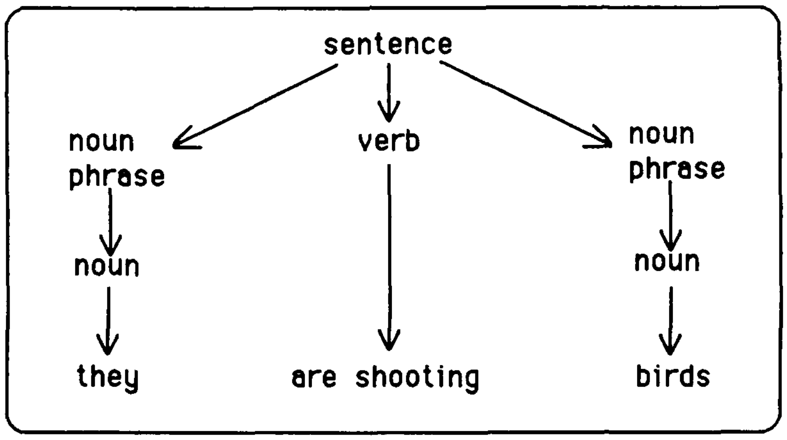


Figure 5. Unambiguous Tree Structure

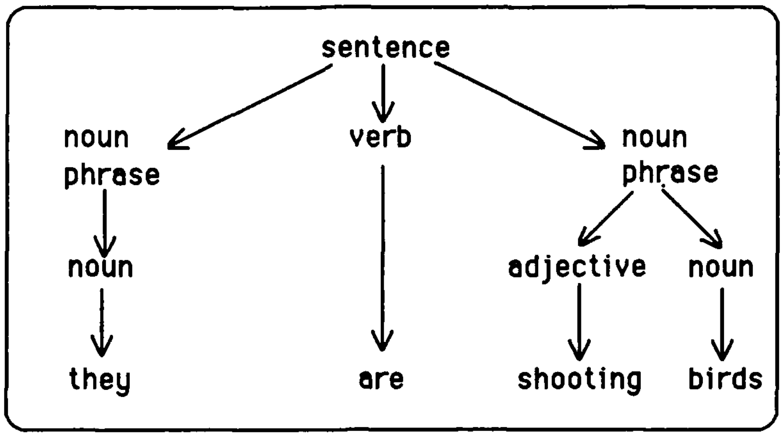


Figure 6. Unambiguous Tree Structure

Because dendrogrammars are equivalent to textual grammars, the dendrogrammars used in Chapter 6 will be defined to operate in the same manner as classical textual grammars to include the use of rewrite rules, productions, and reductions.

C. REWRITING SYSTEMS

A grammar is equivalently called a rewriting system (Ref. 22), and a rewrite rule is called a production within a grammar. The rewrite rule is considered to be the replacement of specific symbols within a string by other symbols. The syntactic rules of the system describe which symbols are allowed to be replaced and which symbols are allowed to do the replacing. Note that there is no requirement that the replacement either increase or decrease the size of the overall term.

Gram1 in Figure 7 is an example of rewrite rules for a textual rewrite system. This textual grammar, or series of rewrite rules, creates strings of zero or more "a"s and none or one b where the "a"s must always come before the b. This grammar is redundant in the fact that it could be rewritten in a simplified manner as shown in Figure 8.

O	->	D
D	->	aD
D	->	aB
D	->	a
D	->	b
aB	->	B

where	O is the starting symbol
	D and B are variables
	a and b are terminals

Figure 7. Gram1

D	->	aD
D	->	a
D	->	b

where	D is the starting symbol and solitary variable
	a and b are terminals

Figure 8. Gram2

Note that Gram1 could also be written with a dendrogrammar. This is depicted in Figure 9, Dendrol.

```
                O
O      ->       \
                  D

                a
D      ->       \
                  D

                a
D      ->       \
                  B

D      ->       a

B      ->       b

a
 \     ->       B
   B
```

Figure 9. Dendrol

Although Gram1 and Dendrol generate the same language, the *dendrogrammar* provides spacial separation and a hierarchial ordering which can aid ones ability to understand internal symbol manipulation as grammars become complicated.

Now armed with a general concept of dendrogrammars and rewriting systems, the author will illustrate an approach to handling "undefined objects" within a specification. As stated before, this method of handling "undefined objects" leads to a consistent understanding of the specification and reduces the combinatorial explosion problems which Goeguen encountered when he enumerated every case of possible error conditions.

VI. IDENTIFYING UNDEFINED OBJECTS

A. STACK SPECIFICATION

The stack specification in textual format is depicted in Figure 10. The term mtstk( ) is a nullary operator which returns an empty stack. Additionally there are three operators for stacks: push, pop, and top. The push operator accepts a stack term and a value and returns another stack. This operation will be considered to be logically equivalent to placing a value on the top of a stacked set of zero or more previous values (defined as a stack). The pop operator similarly accepts a stack term and returns another stack. This operation will be considered to be logically equivalent to removing the top term from a stacked set of values. Finally, the top operator accepts a stack and returns a value. The value it returns will be the top term of the set of stacked terms.

```
push (S,v)    ->    S    (1)
pop (S)       ->    S    (2)
top (S)       ->    v    (3)
mtstk( )      ->    S    (4)
```

Figure 10. Textual Stack Specification - S_{text}

The stack specification can equivalently be described with a tree structure as depicted in Figure 11.

```
   push
   /  \          ->     S              (1)
  v    S

   pop
      \          ->     v              (2)
       S

   top
      \          ->     S              (3)
       S

   mtstk( )      ->     S              (4)
```

Figure 11. Tree Structure Stack Specification - S

B. UNDEFINED TERMS

The stack specification has certain characteristics which must be further defined. One of those areas is undefined terms. Thus, Figure 12 describes a dendrogrammar representation of the undefined terms, U_{SP}, within the stack specification.

```
   pop
      \                    (1)
       mtstk( )

   top
      \                    (2)
       mtstk( )
```

Figure 12. U_{SP} - Undefined Terms Within the Stack Specification

Recall that by declaring equations (1) and (2) in U_{SP} as undefined terms one is stating that if such a situation arises in the program/hardware, absolutely nothing is guaranteed about the execution of that system. Methods of resolving an undefined situation is left to the implementation of the specification and therefore, anything from system shutdown to ignoring the expression and continuing processing could occur in actuality.

Note that both terms (1) and (2) from U_{sp} have the term "mtstk( )" within them. Thus, identifying all undefined terms within the stack specification requires recognition of an mtstk( ) term. By recognizing whether a term contains an mtstk( ) expression or subterm, one could then check for undefined terms through pattern matching techniques to recognize the trees (1) and (2) in U_{sp}. But in order to manipulate stack tree structures, one requires a rewrite system (or grammar) which describes the legal process to create (or recognize) a stack tree structure.

C. SPECIFICATION DENDROGRAMMAR

From the specification, S_{text}, we develop the syntactic rules which describe how a stack may be created. This grammar is depicted in Figure 13 and is called T_{sp}. The starting symbol for the grammar is "S." The variables are "A" and "B." Additionally, "v" is any terminal that is being manipulated in the stack term (pushed and popped).

S	->	mtstk( )	
S	->	A mtstk( )	
A	->	A pop	
A	->	A push B	
A	->	epsilon	(null string)
B	->	top S	
B	->	v	

Figure 13. Textual Stack Specification - T_{sp}

Just as a textual grammar was developed from S_{text}, we can develop the syntactic rules which describe how a stack may be created from a dendrogrammar. The stack specification dendrogrammar, D_{sp}, is depicted in Figure 14. The starting symbol for the grammar is T_S. The variables are A and B and "v" is any terminal that is being manipulated in the stack term (pushed and popped).

```
Ts    ->    mtstk( )                    (1)

              A
Ts    ->       \                        (2)
                mtstk

A     ->      A
               \                        (3)
                pop

               A
A     ->        \                       (4)
                 push
                /
               B

A     ->    epsilon      (null string)  (5)

B     ->    top
             |                          (6)
            Ts

B     ->    v                           (7)
```

Figure 14. Stack Specification Dendrogrammar - D_{sp}

In analyzing the operation of a stack, there are distinct points within the stack tree at which the tree may be expanded (by the addition of the terms pop or push). These "expansion points" occur either: 1) prior to a pop or push operator or 2) immediately after a pop or push operator. This observation is consistent with the concept that the terms pop and push must always operate on a stack term (as required by the specification). Thus, the concept of expansion points is implemented in the ambiguous specification grammar, D_{asp}, in Figure 15 and the variable T_{sx} represents the expansion points.

```
Ts    ->    mtstk( )                                  (1)

            Tsx
Ts    ->      \                                       (2)
               mtstk( )

            pop
Tsx   ->      \                                       (3)
               Tsx

            Tsx
Tsx   ->      \                                       (4)
               pop

            push
Tsx   ->    /   \                                     (5)
          Tv    Tsx

            Tsx
Tsx   ->      \
               push                                   (6)
               /
            Tv

Tsx   ->    epsilon        (null string)              (7)

            top
Tv    ->     |                                        (8)
            Ts

Tv    ->    v                                         (9)
```

Figure 15. Ambiguous Stack Specification Grammar - D_{asp}

Note that D_{asp} is an ambiguous grammar by the fact that it could be reduced to D_{sp}. (An ambiguous grammar is a grammar which allows a well formed sentence to be parsed in more than one way (Ref. 23).) The grammar D_{asp} will be used to prove the validity of a grammar which recognizes stack terms logically equivalent to empty stack terms (mtstk( )). The ambiguous grammar allows a sufficient degree of freedom to permit pattern matching techniques require in the proof.

D. SPECIFICATION AXIOMS

Next, axioms are defined to give the stack specification the logical properties desired by the designer of the specification. Figure 16 describes the required stack axioms (D_a).

```
G                        G
 \                        \
  Tsx          -->         pop             (1)
    \                        \
     S                        push
                              /   \
                            Tv     S


 top
  |
 push          <--->       v               (2)
 /  \
v    S
```

Figure 16. Axioms - D_a

Since the concern is to recognize mtstk( ) terms, the rewrite equations dealing with stack values, rewrite equation (2) in D_a , and rewrite equations (8) and (9) in D_{asp} will now be ignored. This is done with no loss of generality since the stack values and their associated axiom do not affect the structure of a stack and it is the structure "mtstk( )" that is of concern - not the values in the stack.

E. AN MTSTK( ) GRAMMAR

Because the goal is to recognize stack terms logically equivalent to mtstk( ), an mtstk( ) grammar is presented in Figure 17. Again note that this is an ambiguous grammar. Also note that the T_V variable is disregarded in the present grammar, D_{mt}.

```
Tmt   ->    mtstk                              (1)

            Tsx
Tmt   ->     \                                 (2)
              mtstk( )

            Tx
             \
              pop
Tx    ->         \                             (3)
                  Tx
                   \
                    push
                   /    \
                 Tv      Tx

Tx    ->    epsilon      (null string)         (4)
```

Figure 17. MTSTK( ) Grammar - D_{mt}

With the given stack specification (D_{mt} - Figure 15), and the axioms (D_a - Figure 16), recognition of undefined objects within the specification is accomplished by recognition of logically equivalent mtstk( ) terms through the use of the empty stack grammar, D_{mt}. Thus, we must show that D_{mt} properly recognizes all mtstk( ) terms.

F. PROVING D_{MT} VALIDITY

The following proof is presented to show the consistency of D_{mt} in properly recognizing all terms that are logically equivalent to mtstk( ).

1. Overview of the Proof

Goal: Prove that any term provably equal to mtstk( ) is generated by D_{mt} grammar (the empty stack grammar)

In order to show that any term provably equal to mtstk( ) (according to the specification and axioms) is generated by D_{mt}, the following two cases must be addressed:

case 1: Any term provably equal to mtstk( ) is generated by D_{mt}.

case 2: The grammar only generates terms provably equal to mtstk( ).

Case 1 of the proof will be accomplished by induction on the number of times the axiom, D_a (1) has been applied to the originating term (starting symbol) mtstk( ). Case 2 of the proof will be a proof by contradiction.

2. Case 1. Any Term Provably Equal to mtstk( ) is Generated by D_{MT}.

a. Base Step:

Let n be the number of times that D_a (1), the specification dendrogrammar axiom, is applied to the originating term of mtstk( ).

If $n = 0$, we are left with he originating term mtstk( ) from the specification dendrogrammar, D_{asp}.

By application of rewrite rule (1) in D_{mt}, mtstk( ) is trivially generated.

b. Inductive Step:

The inductive assumption follows:

Any tree which is provably equal to mtstk in n steps is generated by D_{mt}. In other words, if we have P(n) = T (some tree), we are assuming that P(n) is provably equal to mtstk( ) and we must now prove that P(n+1) = T' is generated by Dmt.

We must now consider 2 cases of the axiom application to the tree T (or P(n)):

Case 1a: The axiom, D_a (1), could be applied to T in a manner which expands the tree. In other words, the axiom adds another push and pop to the original tree, T, to generate T'.

Case 1b: The axiom, D_a (1), could be applied to T in a manner which reduces the tree. In other words, the axiom removes a push and pop from the original tree, T, to generate T'.

Inductive Step 1a. Expansion of the Original Tree.

As discussed before, there will be expansion points in the specification term, T', where the axiom Da (1) will be applied. Da (1) can be applied at any Tsx (specification expansion point) and the questions to be answered include:

1) Where are the specification expansion points located?

2) Does the dendrogrammar, D_{mt}, Properly recognize the expansion when D_a (1) is applied to one of the points specified above?

By analyzing D_{asp} (the specification dendrogrammar), we notice that T_{sx} occurs in three places:

1) Just prior to the term mtstk( ), as noted in D_{asp} (2)
2) Just prior or immediately after the operator pop, as noted in D_{asp} (3) or (4)
3) Just prior or immediately after the operator push, as noted in D_{asp} (5) or (6)

The above 3 cases are similar in the fact that T_{sx} occurs just prior to any stack term. In number one above, T_{sx} occurs prior to the stack term mtstk( ). Next, note by the original stack specification, S, both the push and the pop operators return terms of type stack. This implies that (3) and (5) in D_{asp} both have placed T_{sx} prior to stack terms. Again, referring to the specification, S, we note that pop operates on a stack term and therefore, T_{sx}, in rewrite rule D_{asp} (4) is placed immediately prior to a stack term. Finally, by rewrite rule (1) in S, we note that push operates on both a stack term and a value. T_v is a value and therefore T_{sx} is once again placed immediately prior to a stack term.

If D_{mt} did not allow for expansion for a term at one of the points which are recognized for expansion in D_{asp}, then D_{mt} would not be able to recognize all D_{asp} terms which are equivalent to mtstk( ). We have already qualified each T_{sx} point to occur prior to any stack term and therefore have the requirement that T_x (the mtstk( ) then dendrogrammar expansion point) must occur prior to any (and all) stack terms.

To prove that T_x occurs prior to all stack terms in Dmt we will look at all possible cases where T_x can occur. By rewrite rule (2) in D_{mt}, T_x occurs prior to mtstk( ). The only transition allowed after rule (2) in D_{mt} are (3) and (4). If we strictly apply transition (3) we will note that each T_x expansion is depicted in Figure 18.

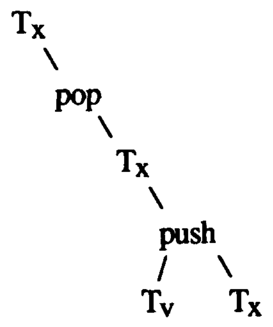


Figure 18. T_x Expression

From this expansion, there is not a stack term (designated by a point prior or after the operators push and pop) that does not have a T_x prior to it.

Next, if we apply transition (4), we are only removing T_x from the tree structure and therefore not adding operators. If no operators are added, no additional stack terms could be created. Thus, this case is moot when considering whether we have a T_x prior to every stack term.

With T_x occurring every where that T_{sp} may occur, our next concern is whether the dendrogrammar D_{mt}, properly recognizes the expansion of T_{sp} by the axiom D_a (1). The axiom D_a (1) is a rewrite rule where T_{sp} is replaced with Figure 19.

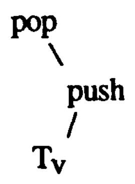


Figure 19. Axiom D_a (1) Replacement

Thus, we need to show that the transition in Figure 20 is equivalent to the transition in Figure 21.

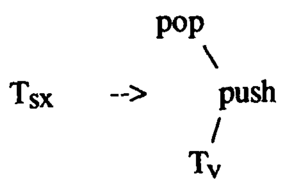


Figure 20. T_{sx} Transition

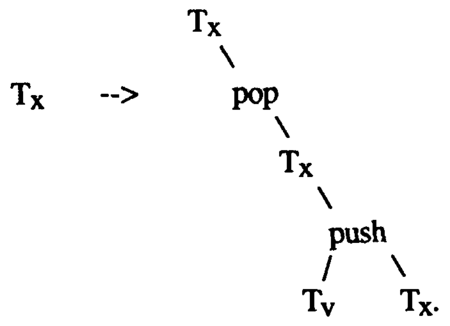


Figure 21. T_x Transition

This is clearly seen through the use of D_{mt} (4) following the initial transition of D_{mt} (3) as observed in Figure 22.

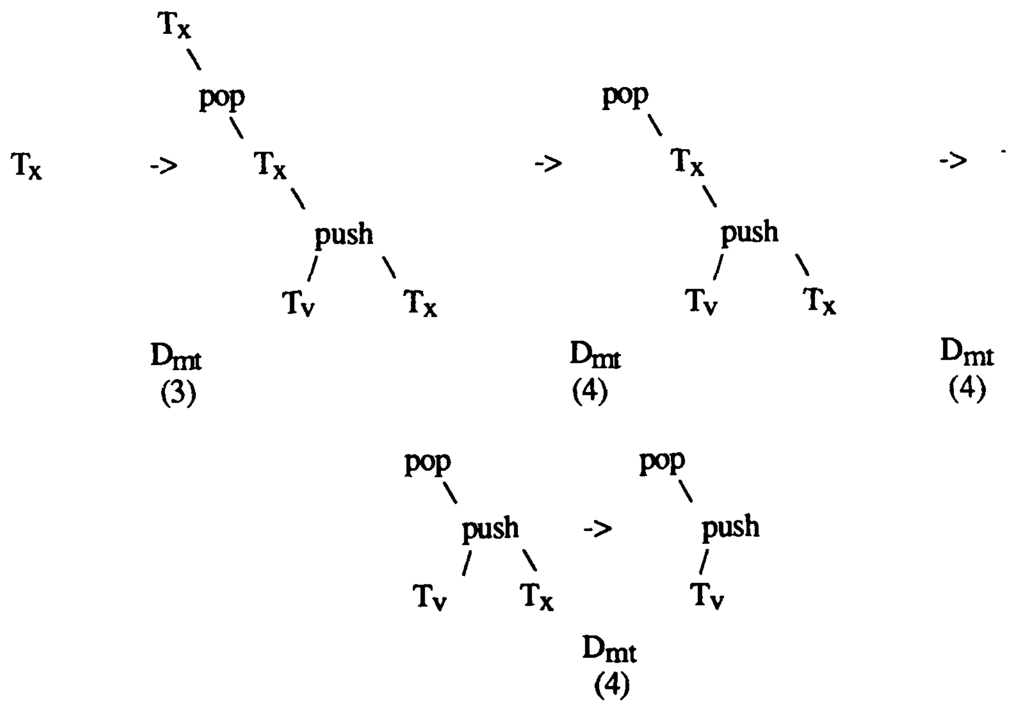

T_x -> T_x pop T_x push T_v T_x
D_{mt} (3)
-> pop T_x push T_v T_x
D_{mt} (4)
->
D_{mt} (4)
pop push T_v T_x -> pop push T_v
D_{mt} (4)

Figure 22. Reduction of T_x

Therefore a transition of T_{sx} of the axiom is equivalent to a transition of T_x by rewrite rule (3) of D_{mt} and then three transition by rewrite rule (4) of D_{mt}.

Inductive Step 1b. Reduction of the Original Tree.

Next we consider what happens to T when Da (1) is applied in the opposite direction as depicted in Figure 23.

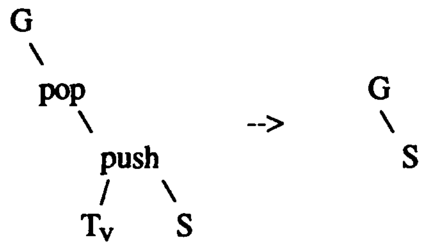


Figure 23. A Tree Reduction

To show that the reduction case is also recognized by the dendrogrammar D_{mt}, we allow only rules (1), (2), and (3) of D_{mt} *to be applied* during expansion of the tree, T_{mt}. This does not affect the generality of the proof because D_{mt} (4) only eliminates the expansion points, T_x, from the tree.

In the specification grammar, the tree created prior to application of the axiom is defined as T_{sp} (and after the axiom is T_{sp}'). Additionally, recall that we are assuming that T_{sp} was created by n steps of the axiom D_a (1) and T_{sp} is semantically equivalent to T_{mt}.

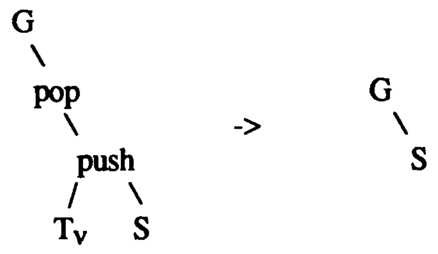
Since our dendrogrammar rules are rewrite rules, we are using a substitution or "tree replacement" (expansion/contraction) concept implementation of the grammar. Because our dendrogrammar D_{mt} will recognize T_{sp} (T_{sp} is equivalent to T_{mt}), and all previous expansions to this point have been provably recognized by D_{mt}, any reduction of the tree, T_{sp}, will place the new stack term,

T_{sp}', at a previous stack term state which equates to one of the previously "recognized" states by D_{mt}. Therefore, any reduction is equivalent to not accomplishing the previously recognized (by D_{mt}) expansion of the stack term T_{sp}' which is equivalent to mtstk( ).

3. **Case 2. The Grammar Only Generates Terms Provably Equal to mtstk( ).**

This will be a proof by contradiction:

Assume the grammar generates a term other than one provably equal to mtstk( ). The axiom D_a (1) states that we can always remove any pair of pop-push operators in the stack term S as noted in Figure 24.



where: G is any combination of pop and push operators (to include the null set)

S is any stack term

Figure 24. Axiom D_a (1)

We can use this axiom to reduce the original stack tree into a "smallest stack tree" (minimized stack term). After we have eliminated all pop-push combination, we are left with four subcases of stacks:

1) We have only pop operators in the stack term (which also includes the nullary operator mtstk( )).
2) We have only push operators in the stack term (which also includes the nullary operator mtstk( )).

3) We are left with only the terminal stack - mtstk( ).

4) We have some series of operators where the push operators come before the pop operators and we are therefore "popping" an mtstk( ).

In analysis, subcases one and two degenerate to the same argument and therefore only one proof for both subcases is presented. Subcase 1 will be specifically proven.

Subcases 1 and 2:

Let S be any stack term which is not provably equal to mtstk( ) and yet was generated by D_{mt}.

If D_{mt} generated S, and S has no push operators to match the multiple pop operators, then D_{mt} must have at least one production (rewrite rule) which will add a pop to the stack term without adding a push. Note that rewrite rule (3) is the only rule that adds a pop to the stack term: however; it also adds a push to the stack term. Therefore, there is no rule which adds a pop operator to the stack term without adding a push operator -- a contradiction of the original assumption.

Subcase 3:

In this situation we can see that D_{mt} (1) generates the terminal mtstk( ) -- which is also a contradiction of the original assumption.

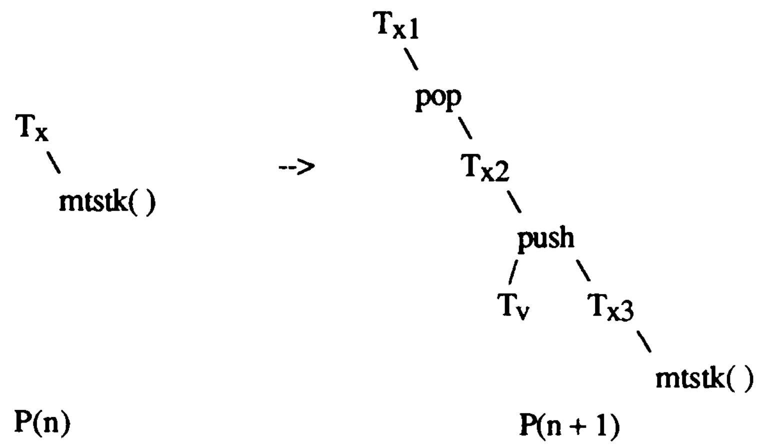
Subcase 4:

If our dendrogrammar is to generate a stack term where the push operators come before the pop operators, then the dendrogrammar, D_{mt}, must be able to position a pop operator directly in front of the terminal mtstk( ). This is only possibility because any pop operator prior to a push operator would be reduced from the stack tree.

By analyzing the D_{mt} dendrogrammar, we see that the only variable that precedes the terminal mtstk( ) is T_x. Now note that the T_x variable has two possible rewrite rules, rewrite rule D_{mt} (4) and rewrite rule D_{mt} (3).

If rule (4) is applied to the variable T_x, we could not achieve an operator (specifically a pop) prior to the terminal mtstk( ).

If rule (3) is applied to the variable T_x, the transformation of Figure 25 occurs.



note: The T_x terms in P(n + 1) are numbered for discussion purposes only and equate to the variable T_x.

Figure 25. T_x Transition by D_{mt} (3)

Thus we can now apply the D_{mt} rewrite rules to T_{x3} in P(n + 1). If T_{x3} is converted by rewrite rule D_{mt} (4) we end up with a push just prior to the term mtstk( ) which is contradictory to the original assumption. Additionally, if T_x is converted by rewrite rule D_{mt} (3) then another T_x is placed just prior to the term mtstk( ). By a simple inductive proof one can see that either a T_x variable will be placed prior to mtstk( ) or the T_x variable is replaced with epsilon causing a push operator to be placed prior to the term mtstk( ) (which again contradicts the assumption that a pop operator is before the mtstk( ) term). Thus both of these possibilities result in a contradiction of the original assumption.

With all four subcases contradicting the original assumption that D_{mt} can generate some term other than a term equivalent to mtstk( ), we have shown that D_{mt} can only generate terms equivalent to mtstk( ).

VII. APPLICATION OF THE EMPTY STACK GRAMMAR

Once we have a grammar which syntacticly identifies terms which lead to undefined objects in the specification, we now identify the actual undefined objects. Referring to U_{sp}, the two terms identified as undefined objects are reiterated in Figure 26.

```
pop
  \                    (1)
   mtstk( )

top
  \                    (2)
   mtstk( )
```

Figure 26. Undefined Objects Within the Stack Specification

Notice that D_{mt} identifies all mtstk( ) terms. Thus, we use a rewrite system (grammar) to identify all undefined objects through the use of pattern matching techniques (parsing). Now it becomes a simple matter of looking for a pop or a top operator prior to any mtstk( ) term. This is similar to finding undefined objects within the real number specification referred to earlier. For example, if divide by zero is undefined, then look for zero terms and check if they are in the denominator of any division equation.

Once the undefined objects are identified, then, using denotational semantics, we can place meaning on this class of objects by declaring them as "don't care" situations.

VIII. A PROCEDURE FOR HANDLING UNDEFINED OBJECTS WITHIN A SPECIFICATION

The following procedure is a summary of the steps which can be used to identify undefined objects within a specification:

1) Define the specification.
2) Define specification axioms which provide the meaning desired from the specification.
3) Specify basic undefined terms within the specification.
4) Create a grammar to identify "undefined objects".
5) Include the "undefined object" grammar as part of the specification and declare these objects as "don't care" situations.

Note: the rewrite rules of the "undefined object" grammar will be used to properly recognize and classify "undefined objects" within the specification.

IX. CONCLUSIONS

Objects which have no interpretation from the syntactic domain into the semantic domain are treated as "undefined objects" in order to avoid the combinational explosion problems encountered when treating these objects as error conditions. "Undefined objects" are considered to encompass both error conditions and "don't care" conditions. "Don't care" conditions are conditions which, if they are implemented, their implementation does not affect the consistency of the specification.

By declaring a class of "undefined objects" in the specification we avoid having to enumerate each specific situation where a syntactic term does not map into a semantic meaning. This class of terms is then uniformly treated when proving the automation of the specification. Therefore, accomplishing automation of the specification phase is more probable.

Although this procedure will work for a class of formal specifications, how broad a range of specifications this procedure will apply to is yet known. Areas of future study might attempt to characterize properties of specifications to which this procedure applies.

LIST OF REFERENCES

1. Yurchak, J. M., The Formal Specification of an Abstract Machine: Design and Implementation, Master Thesis, Naval Postgraduate School, Monterey, California, December 1984.

2. Griffin, R. An Algorithm to Test For Confluence in a System of Left to Right Rewrite Rules, Master Thesis, Naval Postgraduate School, Monterey, California, December 1984.

3. Lilly, N. L., An Algebraic Specification Language for a Syntax Direct Editor, Master Thesis, Naval Postgraduate School, Monterey, California, December 1984.

4. Ozkan, U., A Survey of Properties of Relations Which Have the Confluence Property, Master Thesis, Naval Postgraduate School, Monterey, California, June 1985.

5. Davis, Daniel, Naval Postgraduate School, Monterey, California, Interview, 11 May 1988.

6. Bradley, Professor Gordon H., "Software Engineering Manuscript," p. 4.8, Fall 1987.

7. Bradley, Professor Gordon H., "Software Engineering Manuscript," p. 4.8, Fall 1987.

8. Bradley, Professor Gordon H., Software "Engineering Manuscript," p. 4.30, Fall 1987.

9. MacLennan, Bruce J., Principles of Programming Languages, 2nd ed. p.547, CBS College Publishing, New York, 1987.

10. Davis, Daniel, Naval Postgraduate School, Monterey, California, Interview, 27 January 1988.

11. Bradley, Professor Gordon H., "Software Engineering Manuscript," p. 4.31, Fall 1987.

12. Bradley, Professor Gordon H., "Software Engineering Manuscript," p. 4.33, Fall 1987.

13. Fairley, Richard, Software Engineering Concepts, p.50, McGraw Hill, New York, 1985.

14. Fairley, Richard, Software Engineering Concepts, p.48, McGraw Hill, New York, 1985.

15. Bradley, Professor Gordon H., "Software Engineering Manuscript," p. 4.33, Fall 1987.

16. Beckman, Frank S., Mathematical Foundations of Programming, p. 288, Addison-Wesley Publishing Company, Reading, Massachusetts, 1980.

17. Davis, Daniel, "Advanced Language Topics Notes," Naval Postgraduate School, Monterey, California, 1988.

18. Floyd, R.W., Assigning Meaning to Programs, Proceedings of Sym. Applied Mathematics Math Aspects of Computer Science AMS, Providence, Rhode Island, 1967.

19. Pollack, S. V. and Sterling, T. D., Computing and Computer Science : A First Course with PL/I, pp. 316-319, The MacMillan Company, Toronto, Canada, 1970.

20. Goguen, J. A., and others, An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types, Current Trends in Programming Methodology IV, Data Structuring, R. T. Yeh, ed., Prentice-Hall, 1978.

21. Davis, Daniel, Naval Postgraduate School, Monterey, California, Interview, 11 May 1988.

22. Beckman, Frank S., Mathematical Foundations of Programming, p. 298, Addison-Wesley Publishing Company, Reading, Massachusetts, 1980.

23. Beckman, Frank S., Mathematical Foundations of Programming, p. 288, Addison-Wesley Publishing Company, Reading, Massachusetts, 1980.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1
4.	Computer Technology Programs, Code 37 Naval Postgraduate School Monterey, California 93943-5000	1
5.	Daniel Davis, Code 52vv Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	5
6.	Douglas R. Lengenfelder 18620 32nd. Ave. S. Seattle, Washington 98188	10
7.	AFIT/CIRD Wright - Patterson Air Force Base, Ohio 45433-6583	1
8.	AFIT/NR Wright - Patterson AFB, Ohio 45433-6583	1